

AD A095569

LEVEL ~~II~~

①



⑥

PROCEEDINGS OF THE

ADA DEBUT, Washington, DC,
4-5 September 1988.

APPROVED FOR PUBLIC RELEASE
DISTRIBUTION UNLIMITED

⑫ 127

DTIC
ELECTE
FEB 27 1981
S A D

⑪

SEPTEMBER 1988

JOB

390966

81 2 27 038

DOC FILE COPY

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO. <i>AD-AC 95 569</i>	3. RECIPIENT'S CATALOG NUMBER
4. TITLE and Subtitle PROCEEDINGS OF THE ADA DEBUT		5. TYPE OF REPORT & PERIOD COVERED Conference Proceedings
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s)		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Defense Advanced Research Projects Agency 1400 Wilson Blvd. Arlington, VA 22209		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS		12. REPORT DATE Sep 1980
		13. NUMBER OF PAGES 130
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) DARPA/TIO 1400 Wilson Blvd. Arlington, VA 22209		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report) APPROVED FOR PUBLIC RELEASE DISTRIBUTION UNLIMITED		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		
18. SUPPLEMENTARY NOTES		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) ADA Programming Language Computers		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) (U) Conference proceedings of the ADA program held at U.S. Department of Commerce Auditorium, 4-5 September 1980.		

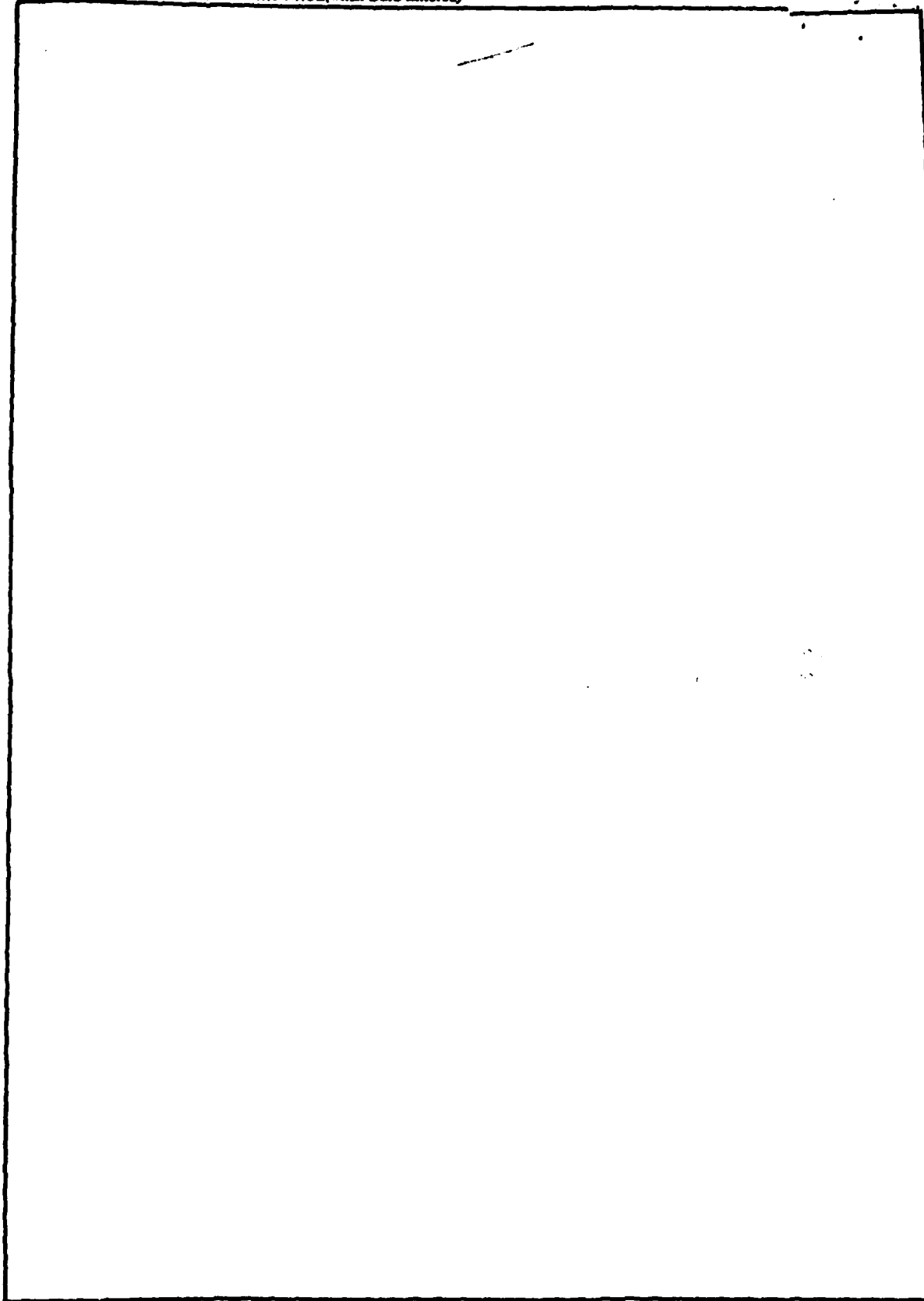
DD FORM 1 JAN 73 1473

EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)



SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

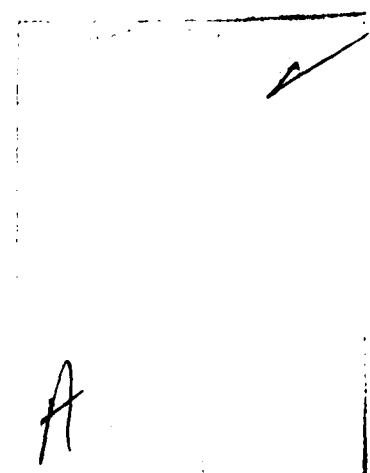
DEBUT OF THE
ADA PROGRAMMING LANGUAGE



4-5 SEPTEMBER 1980

APPROVED FOR PUBLIC RELEASE
DISTRIBUTION UNLIMITED

U.S. DEPARTMENT OF COMMERCE AUDITORIUM



THURSDAY SEPTEMBER 4TH

9:00-10:30	Opening Session	
	Welcome	Dr. Robert R. Fossum Director Defense Advanced Research Projects Agency
	Remarks	Hon William J. Perry Undersecretary of Defense for Research and Engineering
	Introduction to Ada	Dr. Jean D. Ichbiah Principal Language Designer
	Coffee Break	
10:50-12:30	Types	Dr. Ichbiah
	Lunch	
1:45-3:15	Program Structure	Dr. Ichbiah
	Coffee Break	
3:35-5:00	Algorithmic Features	Dr. Ichbiah
	Wine and Cheese Party	
	Hotel Washington	

FRIDAY SEPTEMBER 5TH

9:00-10:30 Tasking Dr. Ichbiah

Coffee Break

10:50-12:30 Exception Handling Dr. Ichbiah
Generic Program Units

Lunch

1:45-2:45 Other Ada Features Dr. Ichbiah

Conclusion

2:45-3:45 Question/Answer Session

3:45 Closing Remarks William E. Carlson
Ada Program Manager

SPEECH
BY
DR. ROBERT R. FOSSUM
DIRECTOR
DEFENSE ADVANCED RESEARCH PROJECTS AGENCY
AT THE ADA DEBUT 4-5 SEPTEMBER, 1980
WASHINGTON, D.C.

I'm really very happy to be here today. I enjoy coming to convocations of this type and seeing old friends, and I'm especially glad to see Bill Whitaker. I can remember working late at night but I never worked any later at night at ARPA than my colleague Bill, when he was in his office working at a terminal on the various things he had to do as Chairman of the High Order Language Working Group.

I asked Bill Perry why in the world he wasn't able to be here this morning and he said, "Well, I have difficulty with accountants; I have a hearing on the Capital Hill that I have to go to". Then he told me a little story related to the accountant syndrome. He said, "Do you know, Bob, about 100 years ago during the Civil War, when R&D was in its infancy in the U.S. Army, the problem of surveillance was severe. The Army was testing a balloon borne observation system up northwest of Washington. A bright young Army officer was the pilot of the balloon, 'Areostat', it was called in those days. He took off one afternoon on a test flight. These balloons are teathered, except on this day the teather broke loose, he drifted through thunder clouds of great magnitude doing his best to land in some sort of controlled crash. He finally succeeded, drifting down and landing on the White House lawn. It turns out that Mr. Lincoln at the time was having a Cabinet meeting. Seeing the balloon come down and crashed, they ran out. The first person to arrive at the gondola of the balloon was a very distinguished looking gentleman. The balloon pilot looked up and said (he was rather dazed), "for goodness sake where am I?" The gentleman answered, "Why you're on a green grassy lawn, of course". This took the pilot by some surprise, and after a few minutes of absorbing that profound statement, he said, "Are you an accountant?" The fellow said, "Why yes, how in the world did you know?" And the pilot replied, "Because the information you have given me is perfectly accurate but utterly useless".

This little story, I think, reflects Bill's frustrations at not being able to be here today, and I bring with me today his best wishes. He is very much interested in this program, as you all know. He is interested in joint programs with NATO, with the Western Alliance, and he feels this language is one of the successes that is coming about in our increasing efforts at joint programs.

I want to extend my personal congratulations and the congratulations of Dr. Perry to Jean Ichbiah and the Ada Design Team for a job well done.

We at ARPA are very happy to have been part of the DoD effort to develop this language. As most of you know, the programming language modernization and convergence effort is coordinated by the High Order Language Working Group, or HOLWG, which includes representatives of all three Services and the Defense Agencies. The German and United Kingdom Ministries of Defense have sent liaison members to the HOLWG. The Services have provided the money for the Ada development, and full time participation by some of their best people.

The Ada program has benefitted from international cooperation beginning with the requirements definition process, and continuing throughout design, test and evaluation. I note that at least nine countries are represented in the audience here today. Experts from the U.S., United Kingdom, Germany, France, and other nations including Japan have contributed to the design effort. By working together, it is my opinion we have developed a much higher quality product for our common use than any of us could have developed. Such cooperative development programs are an effective way to counter what we in the Department of Defense believe (and we believe it is a mutual perception), to be a massive Warsaw Pact investment strategy which constitutes a real threat today.

Achieving more effective armaments cooperation within the Western Alliance is a major goal of this administration, especially the Undersecretary and the Secretary of Defense. The Ada language is certainly one of the significant accomplishments in this major thrust.

Today I would like to share with you my view of the significance of Ada to the Western Alliance, highlight some of the factors which have made this program a success, and summarize DoD's plans for the Ada phase-in period.

WHAT IS THE SIGNIFICANCE OF ADA?

We in this Alliance must take advantage of our lead in various technologies, in particular computer technology, to maintain the balance of power. The Soviets have historically held a quantitative advantage in deployed weapons. In the past, we have had a qualitative advantage, but that advantage is now eroding in the face of massive Eastern Block R&D investments.

These challenges are formidable. But we have some distinct advantages as well: we have a superior technological base, we have a highly competitive industry with greater productivity, and a substantial industrial base necessary to capitalize on the technology base investments.

Our approach is to expedite the development of systems based on technology in which we lead. We have an asymmetric advantage which can produce a distinct military advantage. Two such obvious technologies in the tech base today in this country and all through the west are micro circuit technology and computer technology, which are, of course, closely related and mutually reinforcing.

Military systems of the future will incorporate much more flexible and intelligent control logic. For example, new surveillance systems will have sophisticated algorithms for detecting, identifying, and locating targets; command and control systems will be able to pass target information to fire units very rapidly, essentially in real-time. We will have even more capable precision guided weapons than we have today. We will also be upgrading existing tanks, aircraft and other weapons with intelligent digital control systems to extend their useful life.

Thus, the Ada program has been an effort to provide a high quality common programming language for these critically important military systems of the future.

Our goal, however, has gone beyond merely designing a quality programming language. The usefulness of a language is determined as much by the stability of its specification and the size of its user population as by its detailed technical characteristics.

It takes several years to accumulate a significant collection of software tools to enhance the productivity of programmers using a language. Developers of applications software cannot afford the time or the money to develop large tool kits, so they are for the most part constrained to use existing tools. Quality tool kits are only available for the most popular languages.

Even in very primitive environments with almost no tools, there must be a compiler for every high order language. The benefits of using proven compilers for applications developments are especially compelling. It takes a while for compiler implementors to refine their products, find the most efficient algorithms, and eliminate residual errors.

Our approach to training is also greatly influenced by how widely the programming language is used. With a widely used common language, experienced programmers are available to work on new projects. Hence, development projects do not have to pay for training and for the mistakes which are inevitably made by inexperienced programmers. Furthermore, training can be viewed as a long term investment in future productivity; since during the course of their careers, programmers will work on a variety of systems in the common language.

The goal, then, is to have a language which is both of high technical quality and widely used. The military has been a successful catalyst for new standards in the past, most notably with the COBOL language in the early 1960's. In the Ada effort, we have tried to involve the widest possible community from throughout the Western alliance so that technical requirements of all our real-time and large systems applications would be provided for in the language design.

WHAT ARE THE HIGHLIGHTS OF THE ADA DEVELOPMENT?

The Ada development has been both a modernization and a convergence effort. The research and development community had developed important new software techniques in the late 60's and early 70's which DoD project managers wanted to use. Among these ideas were: structured programming, the ability for users to define their own data types and have compilers enforce type consistency, various high level and structured approaches to parallel processing which reduced the risk of deadlocks and improved system reliability, and the separation of external interfaces from implementation details to enforce modularity. The separate specification of external interfaces is especially important for standard reusable software components.

In 1975 and 1976, each of the Services was proposing to develop a new programming language incorporating these advances. In discussions with our allies, we learned that these pressures were not unique to the U.S. The IFIP Working Group 2.4 and the Long-Term Procedural Language-Europe Group, among others, had already begun the process of consolidating the advances so that a new common and widely adopted standard language could be developed.

Given the benefits from language commonality which we just discussed, we decided to establish a single joint program to develop a common language. The U.K. Ministry of Defense assigned a liaison officer to work full time with us in the United States formulating language requirements.

We could not afford the risk of having this language designed by anyone other than the best designers in the whole world. Who that should be, however, was not immediately obvious. There were several strong candidates.

The solution was to hold an international design competition based on widely reviewed and agreed upon requirements. Our objective has been to have the widest possible participation in the identification of requirements and the review of interim products, but to have a single designer responsible for the internal consistency and integrity of the language design. As you know, the team headed by Jean Ichbiah, of CII Honeywell Bull in Paris in cooperation with the Honeywell Systems and Research Center in Minneapolis won the competition.

The debut today and tomorrow marks the successful completion of extensive test and evaluation of the language design. During the past year, a significant number of applications algorithms have been recoded in Ada. We have received more than 900 language issues, reports and comments. Jean has refined the language based on the results of these inputs, and will be presenting the proposed Ada Standard in a few minutes.

HOW ABOUT THE FUTURE

We are now entering the language phase-in period. A DoD joint project office is being established to maintain the Ada Standard and to coordinate service implementation and training efforts.

We want to begin using Ada in exploratory development applications as soon as possible. Hence, test compilers are being developed for these pilot applications. New York University will deliver an interpreter for the complete Ada language and submit it for validation testing early in 1981. Carnegie-Mellon and Intermetrics are working together to deliver a complete test compiler by the summer of 1981.

The Army and Air Force have already begun to develop advanced development models for Ada compilers. The Army has chosen SofTech to develop its compiler, and they have started work. The Army has a particularly pressing need to begin using Ada to develop battlefield systems, so everything possible is being done to complete an implementation with code generators for the battlefield computers by mid-1982. Capabilities of the existing operating system environment are being used to expedite the development of the required minimal set of programming tools.

The Air Force effort aims to produce a compiler which is embedded in a self-contained, vendor independent, portable environment in accordance with DoD's STONEMAN specification. There will be a competitive design phase involving two or three contractors who are being selected at the present time. The designs will be delivered in Spring 1981 and the system in 1983.

Both the Air Force and Army compilers will be written in Ada, and will be delivered with the essential programming tools. When the initial designs are delivered to DoD in Spring 1981, DoD will select and specify standard interfaces between the environment and the programming tools. Contracts will then be renegotiated to insure compliance with the interface standards, and to achieve the maximum amount of practical commonality between Army and Air Force efforts.

The Navy is waiting for these activities to stabilize before developing a compiler which is unique to Navy systems. They expect to make substantial use of software developed by the Army and Air Force.

Elsewhere, there are a number of compilers already under development in Europe, Japan, and as proprietary ventures by industry. In particular, the German Ministry of Defense, the United Kingdom Ministry of Defense and the Council of the European Communities have compiler procurements already underway. We in DoD will coordinate closely with these efforts.

At the present time, we are assuming that Ada will be used in a variety of diverse environments, and that our goal in standardizing internal interfaces for initial DoD compilers and tools is simply to make the most efficient use of our development resources.

We intend to establish policies which encourage the private sector to innovate in Ada software and in computer hardware to run Ada programs. An important step towards that end is to provide the stability necessary for private industry investment and non-defense use of the language.

The proposed standard document is being reviewed under American National Standards Institute canvass ballot procedures, and we hope that there will be an ANSI standard for Ada in the near future. ANSI will be submitting the Ada standard to the International Standards Organization to establish an international standard for the language.

We intend to take a variety of steps to encourage the widespread use of Ada. The compiler validation test sets will be available to other governments and standards organizations, with appropriate safeguards to insure that the standard is rigidly and uniformly enforced. Source code and documentation for the initial DoD developed compilers will be made available, to provide an early baseline capability. The contracts have been written so that DoD can give the software to anyone in the world, and the recipients will be free to use it for commercial as well as defense purposes.

As soon as possible, we would like to begin competitively procuring Ada compilers and software tools as products, rather than funding their development on a cost reimbursable basis. For example, we will be using Ada programs as benchmarks when buying commercial hardware so that the performance competition is based on the combined performance of the hardware and the compiler. The Ada compiler validation policies will allow Ada interfaces and special functions to be implemented in hardware, and we will encourage such hardware optimizations for some applications.

These steps are part of DoD's overall strategy of using commercial products wherever possible for defense, so that we can benefit from the industrial base that I described earlier and the economies of scale associated with participating in such a large commercial market place.

CONCLUSION

In summary, I would like to emphasize that the Ada language is extremely important to the U.S. Department of Defense and the Western Alliance. In our opinion, Ada will be used to develop our most important systems in the future. We expect Ada will be widely used throughout the U.S. and the European countries for commercial use as well as defense software, and that the Department of Defense in the U.S. will benefit significantly from the resulting availability of trained personnel and privately developed software.

Dr. Perry shares these views, and very much wishes that he could be with us today. He asked me to emphasize his strong support for this program and his desire to see Ada rapidly implemented and used throughout the Services.

Thank you very much ladies and gentlemen. It is indeed a pleasure to be with you today.

VIEWGRAPH'S

FOR

JEAN ICHBIAH'S PRESENTATION

A SIMPLE PROGRAM

- A simple program
- Textual structure
- Lexical structure
- Execution
- Visibility and Separate Compilation
- Relation to other topics

A SIMPLE PROGRAM

```
with SIMPLE_IO;  
procedure COUNT_YOUR_CHANGE is  
  use SIMPLE_IO;  
  
  -- This program reads 6 integers  
  -- and prints the total value  
  COUNT, CHANGE, CENTS, DOLLARS : INTEGER;  
  VALUE : constant array (1..6) of INTEGER :=  
    (01, 05, 10, 25, 50, 100);  
  
  begin  
    CHANGE := 0;  
    for N in 1..6 loop  
      GET(COUNT);  
      CHANGE := CHANGE + VALUE(N) * COUNT;  
    end loop;  
  
    if CHANGE = 0 then  
      PUT("NO CHANGE");  
    else  
      DOLLARS := CHANGE / 100;  
      CENTS := CHANGE mod 100;  
      PUT("DOLLARS : "); PUT(DOLLARS);  
      PUT("CENTS : "); PUT(CENTS);  
    end if;  
  end COUNT_YOUR_CHANGE;
```


TEXTUAL STRUCTURE

```
with SIMPLE-10;  
procedure COUNT-YOUR-CHANGE is  
  ...  
  
  begin  
    ...  
    for ... loop  
      ...  
    end loop;  
  
    if ... then  
      ...  
    else  
      ...  
    end if;  
    ...  
  end COUNT-YOUR-CHANGE;
```

EXECUTION OF A PROCEDURE

procedure COUNT_YOUR_CHANGE is

COUNTS, CHANGE,
CENTS, DOLLARS : INTEGER;

elaborate

declarative
part

begin

CHANGE := 0;

...

CENTS := CHANGE mod 100;

execute

sequence of
statements

end COUNT_YOUR_CHANGE ;

elaborate declarations :

give an existence to corresponding objects

VISIBILITY and SEPARATE COMPILATION

a main program :

with SIMPLE_10;
procedure COUNT.YOUR_CHANGE is

use SIMPLE_10;
COUNT, CHANGE,
CENTS, DOLLARS : INTEGER;

begin

CHANGE := 0;

GET(COUNT);

PUT("NO CHANGE");

CENTS := CHANGE mod 100;

end COUNT.YOUR_CHANGE;

- predefined identifiers
- identifiers from the package
- local identifiers

POINTS RAISED BY THE EXAMPLE

DECLARATIVE PART

object declarations

types

type declarations

SEQUENCE OF STATEMENTS

expressions

statements

PACKAGES and COMPILATION UNITS

VISIBILITY

Points not raised

TASKING, GENERIC UNITS,
EXCEPTIONS, REPRESENTATION

The way by which we impose
structure on data objects

- Need to describe objects
Factorization of properties
maintainability
- Need to be able to say something
about the properties of objects
readability
- Need to guarantee that properties
of objects are not violated
reliability
- Need to hide implementation
details

TYPES

MOTIVATION FOR TYPES

WHAT IS A TYPE

ENUMERATION TYPES

CONSTRAINTS and SUBTYPES

NUMERIC TYPES

ARRAY TYPES

RECORD TYPES

ACCESS TYPES

TYPE EQUIVALENCE

WHAT IS A TYPE

A type characterizes :

- a set of values
- a set of operations
applicable to the values

WHAT IS A TYPE

Classes of types :

scalar types values have no components

- enumeration types
- numeric types

composite types values consist of component values

- array types
- record types

access types values provide access to other objects

private types values are not known to the users : only the set of operations is known

* RULES

All objects must be declared

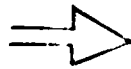
objects can be variables
 constants

Each object gets a type in
its declaration

step 1

Any operation on an object
must preserve its type

step 2



The type of any object is invariant
during program execution :
it is the type given in the
object declaration

CHARACTERISTICS OF ENUMERATION TYPES

type DAY is (MON, TUE, WED, THU, FRI, SAT, SUN);

Set of values:

MON TUE WED THU FRI SAT SUN

Set of operations applicable to these values:

= /= test for (in)equality

> >= < <= test for order

MON < FRI

if TODAY >= TUE then

:= assignment

attributes { DAY'FIRST -- MON
DAY'LAST -- SUN
DAY'SUCC(D) -- e.g. DAY'SUCC(MON) = TUE

ENUMERATION TYPES

enumeration type declarations:

type DIRECTION is (NORTH, EAST, SOUTH, WEST);

type DAY is (MON, TUE, WED, THU, FRI, SAT, SUN);

object declarations:

GOAL : DIRECTION;

TODAY : DAY;

MIDDLE : constant DAY := THU;

correct assignments:

GOAL := WEST;

TODAY := MON;

TODAY := MIDDLE;

incorrect assignments:

GOAL := SUN;

TODAY := 8;

GOAL := TODAY;

GOAL := GOAL + NORTH;

MIDDLE := WED;

} compiler
detectable

EXAMPLES

type definition :

(MON, TUE, WED, THU, FRI, SAT, SUN)

type declaration :

type DAY is (MON, TUE, WED, THU, FRI, SAT, SUN);

object declarations :

```
TODAY : DAY;           -- a variable  
START  : DAY := MON;    -- initialized  
MIDDLE : constant DAY := THU; -- a constant  
YESTERDAY, TOMORROW : DAY; -- two variables
```

OTHER FORMS of ENUMERATION TYPES

Character types

type DIGIT is

('0', '1', '2', '3', '4',
'5', '6', '7', '8', '9');

type HEX_LETTER is ('A', 'B', 'C', 'D', 'E', 'F');

type MIXT is (START, 'A', STOP, 'B');

enumeration literals can be either
identifiers or character literals

CONSTRAINTS

- a constraint restricts the set of possible values of a type
- a constraint does not change the set of applicable operations

range constraints :

D : DAY range MON .. FRI;

L : DAY range THU .. FRI;

range lower_bound .. upper_bound

SUBTYPES

motivation : factorization for
readability and maintainability

D :	DAY <u>range</u> MON .. FRI ;
E :	DAY <u>range</u> MON .. FRI ;
F :	DAY <u>range</u> MON .. FRI ;

a
subtype
indication

factorize by a subtype :

subtype WEEKDAY is DAY range MON .. FRI ;

D : WEEKDAY ;	}	equivalent to the former declaration
E : WEEKDAY ;		
F : WEEKDAY ;		

a subtype name is an abbreviation
for a subtype indication

WHAT HAVE WE SEEN SO FAR ?

- what is a type
- enumeration types
- what is a constraint
- subtypes

Similar notions exist for
other types :

array types : index constraints

record types : discriminant constraints

real types : accuracy constraints

ARRAY TYPES

type VECTOR is

array(INTEGER range <>) of REAL ;

index type(s)

component type

set of values

each VECTOR

- has components of type REAL
- is indexed by values of type INTEGER.
In particular the
index bounds
are of type INTEGER

two VECTORS need not have the
same bounds

EXAMPLES OF ARRAY TYPES

type MATRIX is
array (INTEGER range <>, INTEGER range <>) of REAL;
two-dimensional

subtype NATURAL is INTEGER range 1 .. INTEGER'LAST;

type STRING is array(NATURAL range <>)
of CHARACTER;

strings are arrays of characters
indexed by natural numbers

INDEX CONSTRAINTS

An index constraint serves to specify the index bounds of an array object

V : VECTOR (-10 .. 10);

W : VECTOR (1 .. 1000);

Array subtype

subtype VECT is VECTOR (0 .. N);

INDEX BOUNDS

the index bounds of a constant
are obtained from its initial value

```
MESSAGE : constant STRING :=  
           "how many characters";
```

A formal parameter is constrained by
the index bounds of the corresponding
actual parameter

```
procedure PUT(s : STRING);
```

```
PUT(MESSAGE);
```

ARRAY TYPES

set of operations

$=$ \neq test for (in) equality
 $:=$ assignment possible if
 same number of components
 (but bounds need not
 be the same)

indexing indexed component: $V(I)$

slices for one-dimensional arrays
 $V(I \dots J)$

(a slice is an array value)

notation for array values : aggregates

$A : \text{VECTOR}(1..10) := (1..10 \Rightarrow 0.0);$

$A := (1..3 \Rightarrow 1.5, 4..7 \Rightarrow X, \text{others} \Rightarrow Y);$

$A(2..4) := (1.0, 5.0, 3.0);$

RECORD TYPES

type DATE is

record

MONTH	:	MONTH_NAME;
DAY	:	INTEGER <u>range</u> 1..31;
YEAR	:	INTEGER <u>range</u> 0..3000;

end record;

RECORD TYPES WITH DISCRIMINANTS

type PERSON (SEX : GENDER := F) is

record

```
BIRTH: DATE;  
case SEX is  
  when M => BEARDED: BOOLEAN;  
  when F => CHILDREN: INTEGER;  
end case;  
end record;
```

set of values

triples of the form

(M, (MAR, 25, 1940), FALSE)

{ SEX = M
 BIRTH
 BEARDED

triples of the form

(F, (JAN, 23, 1943), 3)

{ SEX = F
 BIRTH
 CHILDREN

DISCRIMINANTS

set of operations

- discriminants may only be changed by global record assignments

P : PERSON; -- P.SEX = F

P := (M, SOME_DATE, TRUE); -- P.SEX = M

P.SEX := ... -- Illegal !

- access to a component of a variant part depends on the value of the discriminant

P.BEARDED

P.CHILDREN raises CONSTRAINT_ERROR
if P.SEX = M

RECORD SUBTYPES

discriminant constraint

JOHN : PERSON (SEX \Rightarrow M);

MARY : PERSON(F) := (F, (MAR, 3, 1970), 0);

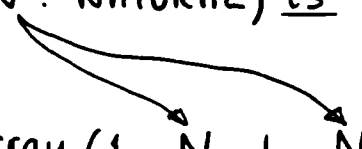
record subtypes

subtype MALE is PERSON (SEX \Rightarrow M);

subtype FEMALE is PERSON (SEX \Rightarrow F);

ANOTHER EXAMPLE

type SQUARE (N : NATURAL) is
record
 MAT : array (1..N, 1..N) of REAL;
end record;



bounds may depend on a discriminant

- An object declaration must give the discriminant value

A, B : SQUARE(8); -- 8 x 8 matrix

- the size of the matrices cannot be changed
- it is impossible to create a rectangular matrix using the type SQUARE

TYPE EQUIVALENCE

when do two objects have the same type?

each type definition defines
a distinct type



each type name denotes a distinct type



two objects have the same type
if their declarations refer
to the same type name

name equivalence.

TYPE EQUIVALENCE

X and Y are not of the same type

X : array (1..10) of INTEGER;

Y : array (1..10) of INTEGER;

two type definitions
hence TWO distinct types

U and V are of the same type

U, V : array (1..10) of INTEGER;

one single type definition

Or better even : name the type

type TABLE_10 is array (1..10) of INTEGER;

A, B : TABLE_10;

C : TABLE_10;

EXPLICIT CONVERSIONS BETWEEN ARRAY TYPES

type VECTOR is array (INTEGER range <>) of REAL;
procedure SORT (X : in out VECTOR);

type TABLE is array (INTEGER range <>) of REAL;
procedure LIST (X : in TABLE);

A : TABLE (1 .. 100);
B : VECTOR (0 .. 1000);
C : array (1 .. 2000) of REAL;

LIST(A); SORT(B);

SORT(VECTOR(A)); LIST(TABLE(B))

SORT(VECTOR(C)); LIST(TABLE(C));

SUMMARY

Notion of type

set of values
set of operations

motivated by reliability

Classes of types

scalar: enumeration
numeric

composite: array
record

access

private types

Constraints and subtypes
for all classes

Name equivalence

SOFTWARE COMPONENTS

M.D. MacIlroy, 1963

"I would like to see components become a dignified branch of software engineering.

I would like to see standard catalogs of routines classified by precision, robustness, time-space requirements and binding time of parameters."

- manufacturers of standard components
- catalogs of components
- parameterized components

GENERAL PROGRAM STRUCTURE

how to organize text
compilation units

scope and visibility
Algol like visibility rules
and extensions

Packages
for better control of visibility
for logical modularity

private types

separate compilation
for physical modularity

methodological impact

PROGRAM UNITS

three forms

subprograms

packages

tasks

program units may be nested

classical Algol-like block structure

program units may be
separately compiled

library units

context specifications

SCOPE AND VISIBILITY

declared identifiers have a scope.

the scope of a declaration is the region of text where the declaration has an effect

different identifiers may be visible at different points of text

the declaration of an entity is visible at a given point if an occurrence of the corresponding identifier at this point denotes the entity

BASIC VISIBILITY RULES

Algol-like rules: direct visibility

- identifier from outer contexts are visible
- an inner identifier may hide an outer homonym

Dot notation : selected component
unit_name.identifier

Named record components in aggregates

Named parameter associations

SELECTED COMPONENTS

procedure OUTER is

A : BOOLEAN;

B : BOOLEAN;

procedure INNER is

B : BOOLEAN; -- hides OUTER.B

C : BOOLEAN;

begin

B := A; -- INNER.B := OUTER.A

C := OUTER.B; -- INNER.C := OUTER.B

end;

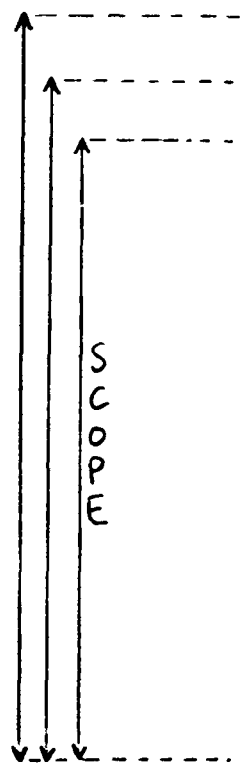
begin

...

end OUTER;

selected components are also
used to denote record components
and items declared in packages

VISIBILITY OF RECORD COMPONENTS



```
procedure P is
  ...
  type DATE is
    record
      MONTH : MONTH_NAME;
      DAY    : INTEGER range 1 .. 31;
      YEAR   : INTEGER range 0 .. 3000;
    end record;

  TODAY : DAY;
begin
  TODAY.DAY := 10;    -- selected
                    -- component

  TODAY :=
    (YEAR => 1980, MONTH => SEP, DAY => 10);
                    -- named aggregate
end P;
```

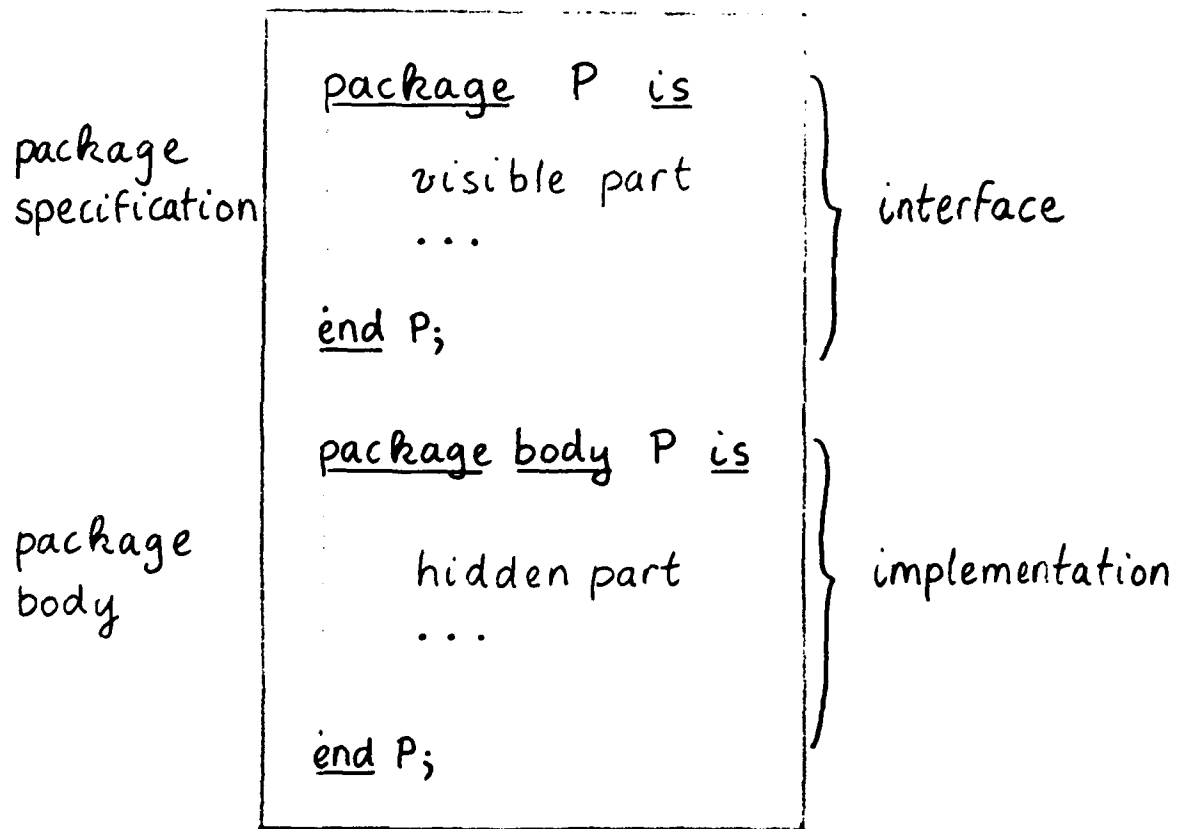
of course, only possible within scope

PACKAGES

Packages are used to formulate

- named collections of declarations
- groups of related subprograms
- private data types

PACKAGE SPECIFICATION AND BODY



A Package

TEXTUAL SEPARATION

procedure specification:

```
procedure SORT ( V : in out VECTOR);
```

procedure body:

```
procedure SORT ( V : in out VECTOR) is  
begin  
    ...  
    ...  
end SORT;
```


FORMS OF PACKAGES

package WORK_DATA is

type DAY is (MON, TUE, WED, THU, FRI, SAT, SUN);

type TIME is delta 0.01 range 0.0 .. 24.0;

type TIME_TABLE is

array (DAY) of TIME;

WORK_HOURS : TIME_TABLE;

NORMAL_HOURS : constant TIME_TABLE :=

(MON .. FRI => 8.0, SAT | SUN => 0.0);

end WORK_DATA;

PACKAGE EXAMPLE : RANDOM

package RANDOM is

type REAL is digits 7;

procedure SET_SEED (START : INTEGER);

function RAND_INT (L, H : INTEGER) return INTEGER;

function UNIFORM (L, H : REAL) return REAL;

end RANDOM;

```

package body RANDOM is
    SEED : INTEGER;
    CO-SEED: REAL;
    -- local data

    procedure CONGRUENCE is
    begin
        SEED := ...; CO-SEED := ...;
    end;
    -- local procedure

    procedure SET-SEED(START: INTEGER) is
    begin
        ...
    end;

    function RAND-INT ... end;
    function UNIFORM ... end;
begin
    SET-SEED(1); -- initialization
end RANDOM;

```

INFORMATION HIDING

Textual separation of the interface

- the visible part defines the logical interface
- the implementation may be both
protected and
physically hidden

PRIVATE TYPES

package SIMPLE_INPUT_OUTPUT is

type FILE is private;

NO_FILE : constant FILE;

procedure ASSIGN (F : out FILE);

procedure READ (F : in FILE; ...);

procedure WRITE (F : in FILE; ...);

private

type FILE is new INTEGER range 0 .. 50;

NO_FILE : constant FILE := 0;

end SIMPLE_INPUT_OUTPUT;

package body SIMPLE_INPUT_OUTPUT is

type FILE_DESCRIPTOR is record ... end record;

DIRECTORY : array (FILE) of FILE_DESCRIPTOR;

procedure ASSIGN (F : out FILE) is

begin

...

end ASSIGN;

...

begin

-- initialization of DIRECTORY

end SIMPLE_INPUT_OUTPUT;

LIMITED PRIVATE TYPES

package INPUT.OUTPUT is

type IN_FILE is limited private;

procedure OPEN (F : in out IN_FILE; ...);

procedure CLOSE (F : in out IN_FILE; ...);

procedure READ (F : in IN_FILE; ...);

FILE_OPEN_ERROR : exception;

private

type IN_FILE is

record

FILE_INDEX : INTEGER := 0;

end record;

end INPUT.OUTPUT;

SUMMARY ON PRIVATE TYPES

- values are not known outside the package
- the only operations available are those of the visible part
- $\left. \begin{array}{l} := \\ = \neq \end{array} \right\} \text{ unless limited}$

ACCESS TO THE VISIBLE PART

dot notation : selected components

```
F : INPUT_OUTPUT.IN_FILE ;
```

```
INPUT_OUTPUT.OPEN ( F ) ;
```

```
INPUT_OUTPUT.READ ( F , X ) ;
```

collectively : use clause

declare

```
    use INPUT_OUTPUT ;
```

```
    F : IN_FILE ;
```

begin

```
    ...
```

```
    OPEN ( F ) ;
```

```
    ...
```

```
    READ ( F , X ) ;
```

end;

SUMMARY ON PACKAGES

uses of packages cover

- named collections of declarations
- groups of subprograms
- private types

textual separation of

- package specification
- package body

packages support information hiding

packages serve to show

private types

- values are not known outside

access to the visible part

- selected component
- use clause

SEPARATE COMPILATION

- same degree of type checking across separately compiled units as within a compilation unit
- supports program construction
 - top down development :
separate compilation of the parts of a program : subunits
 - bottom up development :
reusing program modules : library units
- grain of change :
recompilation of a unit

COMPILATION UNITS

several possible forms

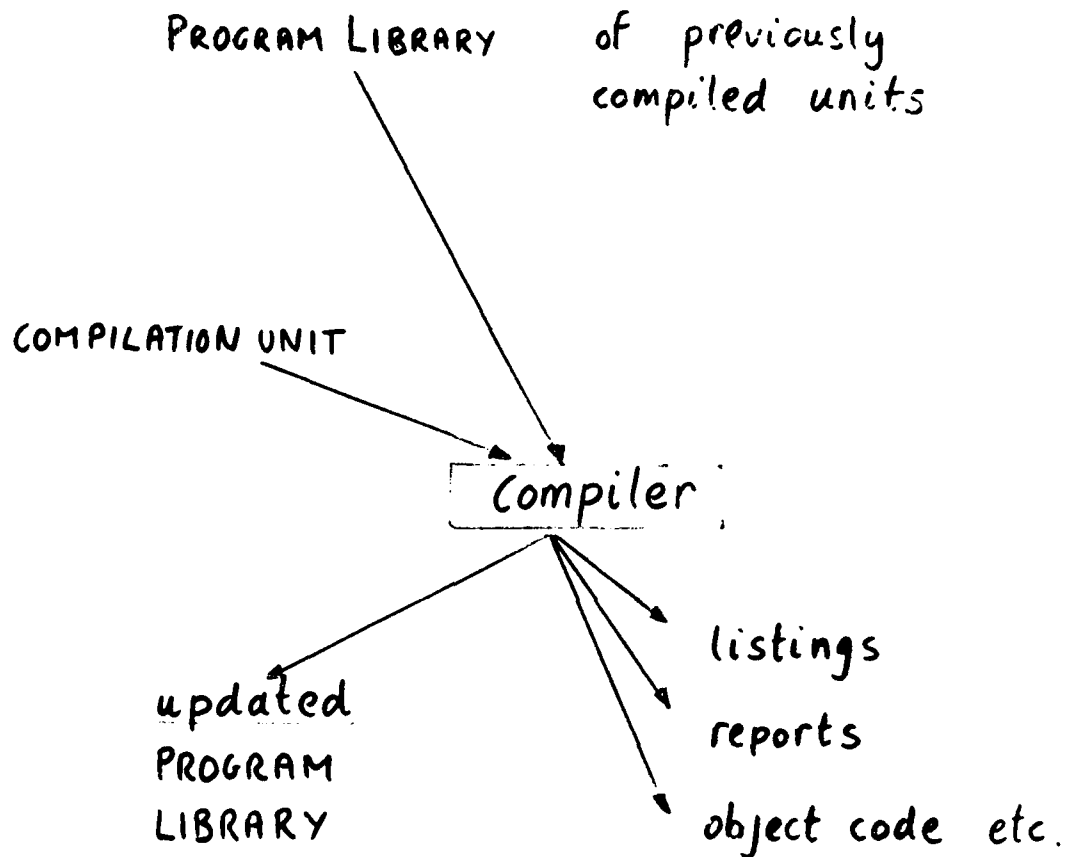
library
units

- package declaration or body
- subprogram declaration or body
- subunits

context specification: with clause

```
with TEXT_IO, REAL_OPERATIONS;  
procedure QUADRATIC_EQUATION is  
    ...  
  
end QUADRATIC_EQUATION;
```

COMPILATION MODEL



either addition of a
new unit or update
of an old unit

"BOTTOM UP" PROGRAM DEVELOPMENT

GENERALLY USABLE PACKAGES

no with
clause

```
package MATH_FUNCTIONS is  
    ...  
end MATH_FUNCTIONS;
```

some
dependence

```
with MATH_CONST;  
package SURVEYING is  
    ...  
end SURVEYING;
```

```
with MATH_CONST, MATH_FUNCTIONS;  
package body SURVEYING is  
    ...  
end SURVEYING;
```

ANY MAIN PROGRAM

"TOP DOWN" PROGRAM DEVELOPMENT

with MATH_FUNCTIONS;

procedure TOP is

use MATH_FUNCTIONS;

type REAL is digits 10;

R, S : REAL;

package D is

PI : constant := 3.14159_26536;

Function F(X : REAL) return REAL;

procedure G(Y, Z : REAL);

end D;

package body D is separate;

BODY

procedure Q(U : REAL) is separate;

STUBS

begin

...

Q(R);

...

D.G(R, S);

...

end TOP;

COMPILATION SUBUNITS

SUBUNITS

A SUBUNIT

```
separate(TOP)
procedure Q(U: REAL) is
  use D;
begin
  ...
  U := F(U);
  ...
end Q;
```

ANOTHER
SUBUNIT

```
with INPUT_OUTPUT;
separate(TOP)
package body D is
  -- local declarations

  function F(X: REAL) return REAL is
    begin
      ...
    end F;
  procedure G(Y,Z: REAL) is separate,
  begin
    ...
  end D;
```

BODY
STUB

NOTE ON VISIBILITY

which identifiers are visible at the start of a compilation unit?

- For a library unit
 - all identifiers of the predefined environment :
the package STANDARD
 - all the names listed in the with clauses
 - the name of the unit itself
with A,B,C;
package P is ... end P;
- For a subunit
same as at the stub, plus with clauses

METHODOLOGICAL IMPACT

- bottom up development

libraries of packages

once the interface is defined,
the body can be redefined independently
of the using units

need for operations on libraries

- top down development

agree on interface then develop subunits

- physical modularity

- physical information hiding

towards an industry of software components

ALGORITHMIC ASPECTS

NAMES and EXPRESSIONS

STATEMENTS

SUBPROGRAMS

procedures, functions,
and operators

overloading

SOME DECLARATIONS

type DATE is

record

MONTH : MONTH_NAME;

DAY : INTEGER range 1 .. 31;

YEAR : INTEGER range 0 .. 3000;

end record;

type PERSON (SEX : GENDER) is

record

BIRTH, DEATH : DATE;

ADDRESS : STRING (1 .. 30);

...

end record;

subtype KING is PERSON(M);

LOUIS : array (1 .. 18) of KING;

JOHN : KING :=

(SEX => M,

BIRTH => (DEC, 24, 1167),

DEATH => (OCT, 18, 1216), ...);

NAMES

identifiers

LOUIS

MONTH_NAME

KING

indexed component

LOUIS(16)

selected component

JOHN.BIRTH

attribute

MONTH_NAME'FIRST -- JAN

MONTH_NAME'LAST -- DEC

function call

HEIR(JOHN)

slice

LOUIS(11 .. 18)

NAMES

Combinations of the previous forms
are also names

LOUIS(11). BIRTH

LOUIS(11). BIRTH. YEAR

JOHN. ADDRESS(12)

LOUIS(14). ADDRESS(1)

HEIR(JOHN). BIRTH. DAY

LOUIS(9 .. 12)(10) -- LOUIS(10)

OPERATORS and EXPRESSIONS

Six precedence levels

lowest	logical operators	<u>and</u> <u>or</u> <u>xor</u> <u>and then</u> <u>or else</u>
	relational operators	= /= < <= > >= <u>in</u> <u>not in</u>
	adding operators	+ - &
	unary operators	+ -
	multiplying operators	* / <u>rem</u> <u>mod</u>
highest	exponentiating operator	**

EVALUATION OF EXPRESSIONS

operators of higher precedence are applied first

parentheses to impose a specific order

the order of evaluation of the two operands of an operator is not defined (except for short circuit control forms)

All operands of an expression are evaluated (unless it contains short circuit control forms)

SUNNY and WARM

NEXT_CAR.OWNER != null and then NEXT_CAR.OWNER.AGE > 25

STATEMENTS

Simple assignment statement
or compound statements

if ... then

...

end if;

case ... is

...

end case;

loop

...

end loop;

also accept and select statements and

declare

...

begin

...

end;

SUBPROGRAMS

Conventional parameterized program units

Two forms of subprograms:

Procedures

a procedure call
is a statement

Functions

- called in expressions
- return a value

Operators are defined as functions

SUBPROGRAM BODIES

procedure identifier formal_part is
 declarative part
begin
 sequence of statements
end identifier ;

function designator formal_part return subtype_indication is
 declarative part
begin
 sequence of statements
end designator ;

a designator is either an identifier or
an operator symbol, for example "*"

SUBPROGRAM DECLARATIONS

subprogram declarations may be introduced for readability considerations

```
procedure QUADRATIC_EQUATION  
    (A, B, C : in REAL;  
     ROOT_1, ROOT_2 : out REAL;  
     OK : out BOOLEAN);
```

```
function INNER (X, Y : VECTOR) return VECTOR;
```

they are necessary in the following cases:

- packages
- mutual recursion

(subprogram specifications also appear in generic declarations and renaming declarations)

PARAMETER MODES

three possible modes

in the parameter acts as a local constant whose value is provided by the corresponding actual parameter.

out the parameter acts as a local variable whose value is assigned to the actual parameter as a result of the execution of the subprogram.

in out the parameter acts as a local variable and permits access and assignment to the actual parameter.

in is the default

For scalar and access types, effect achieved by copy. For array, record, or private types: either by copy or by reference.

FUNCTIONS AND OPERATORS

```
Function INNER(x,y: VECTOR) return VECTOR is  
|  
|   ...  
begin  
|  
|   ...  
|   return SUM;  
end INNER;
```

```
Function "*" (x,y: VECTOR) return VECTOR is  
|  
|   ...  
begin  
|  
|   ...  
|   return SUM;  
end "*";
```

```
A, B : VECTOR(1..N);  
R    : REAL;
```

```
R := INNER(A, B);
```

```
R := A * B;
```

only in parameters

PROCEDURE AND FUNCTION CALLS

POSITIONAL CALLS

INNER(A, B)

QUADRATIC-EQUATIONS(L, M, N, P, Q, STATUS);

NAMED PARAMETER ASSOCIATIONS

formal \Rightarrow actual

- QUADRATIC-EQUATIONS(L, M, N,
ROOT_1 \Rightarrow P, ROOT_2 \Rightarrow Q,
OK \Rightarrow STATUS);
- positional associations must precede
named associations
- the order of named associations
is immaterial

DEFAULT PARAMETERS

A default value may be given for an in parameter in a subprogram specification

```
type STEAK_TYPE is (RARE, MEDIUM, WELL-DONE);  
type POTATO_STYLE is (BAKED, CREAMED, FRIES);  
type DRESSING_KIND is (BLUE-CHEESE,  
    THOUSAND-ISLAND, OIL-AND-VINEGAR, FRENCH);
```

```
procedure ORDER_DINNER
```

```
    (STEAK   : in STEAK_TYPE   := MEDIUM,  
     POTATO  : in POTATO_STYLE := BAKED,  
     DRESSING: in DRESSING_KIND := FRENCH);
```

```
ORDER_DINNER(STEAK => RARE);
```

```
ORDER_DINNER(MEDIUM, CREAMED);
```

```
ORDER_DINNER(RARE, DRESSING => BLUE-CHEESE);
```

```
ORDER_DINNER;
```

OVERLOADING

- Overloading does not hide
- The identification uses the context
- IF the context does not suffice use qualification

procedure REGISTER (D : DAY);
procedure REGISTER (S : OBJECT);

REGISTER (TODAY); } no problem
REGISTER (NEXT_PLANET); } with variables

REGISTER (TUE); } no problem with non-
REGISTER (EARTH); } overloaded literals

REGISTER (SUN); } the solution is
REGISTER (DAY'(SUN)); } to qualify

SUBPROGRAM SUMMARY

Procedures and Functions

Specification may precede body
- must match

Three possible modes
in out inout

Positional and Named calls

Default in parameters

Overloaded subprograms

TASKING CONCEPTS

TASKS and TASK TYPES

task body , specification , objects

TASK EXECUTION

activation termination

COMMUNICATION

entry calls } rendezvous
accept statements }

TIMING

delay

SELECT STATEMENT

multiple wait
timed and conditional calls

SCHEDULING

priorities, interrupt

Classification of tasks

TASK TYPES

task specification

<u>task type</u> T <u>is</u>	{	the interface to other tasks
{entry declaration}		
<u>end</u> T;		

task body

<u>task body</u> T <u>is</u>	{	the actions of the tasks of the type
declarative part		
<u>begin</u>		
sequence of statements		
<u>end</u> T;		

task object declarations

V, W : T;

morphology similar to packages

TASK TYPES

a task type is a limited private type

no assignment

no (in) equality

task objects can be in parameters

an access type can refer to a task type

```
task type KEYBOARDA.DRIVER is  
  entry READ(C : out CHARACTER);  
  entry WRITE(C : in CHARACTER);  
end;
```

```
type KEYBOARD is access KEYBOARD.DRIVER;
```

```
TERMINAL : KEYBOARD := new KEYBOARD.DRIVER;
```

--KEYBOARD is not a limited private type

A SIMPLE EXAMPLE

```
procedure ARRIVE_AT_AIRPORT is
  task CLAIM_BAGGAGE;
  task RENT_A_CAR;

  task body CLAIM_BAGGAGE is
    ...
  end;

  task body RENT_A_CAR is
    ...
  end;

  begin    -- CLAIM_BAGGAGE and RENT_A_CAR
           -- become active

           BOOK_HOTEL;

  end;    -- await termination of
           -- CLAIM_BAGGAGE and RENT_A_CAR
```

COMMUNICATION BETWEEN TASKS

task specification: the interface

```
task LINE_TO_CHAR is  
|   entry PUT_LINE ( L : in LINE);  
|   entry GET_CHAR ( C : out CHARACTER);  
end;
```

task body:

```
task body LINE_TO_CHAR is  
|   BUFFER : LINE;  
begin  
|   -- statements describing the actions  
|   -- performed by the task  
end LINE_TO_CHAR;
```

type LINE is array (1 .. 80) of CHARACTER;

ENTRIES AND ACCEPT STATEMENTS

entry declaration

entry PUT_LINE (L : in LINE);

similar to
a procedure
declaration

entry call

LINE_TO_CHAR.PUT_LINE (MY_LINE);

similar to
a procedure
call

accept statement

accept PUT_LINE (L : in LINE) do
| BUFFER := L;
end PUT_LINE;

similar to
an inline
procedure
body

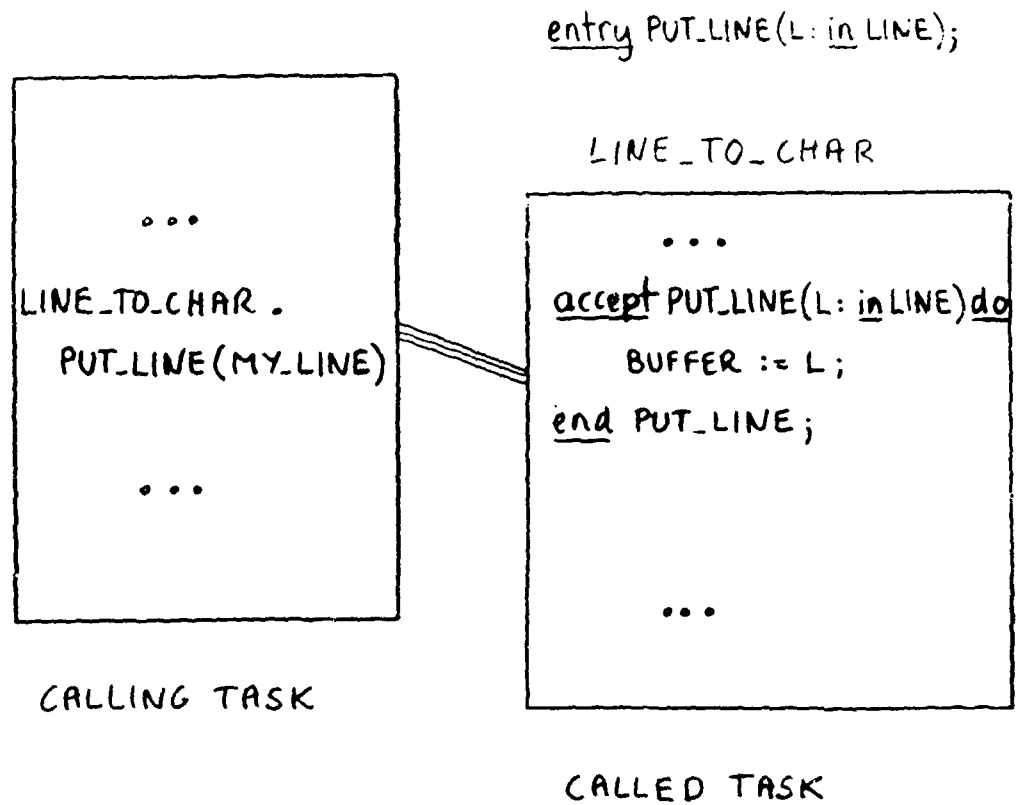
RENDEZVOUS :

between entry call and accept statement

QUEUES :

several tasks may be waiting on an entry

RENDEZVOUS



- whoever is there first waits for the other
- when both are there, the accept is executed
- thereafter, the calling and the called task continue in parallel

A SERVER TASK

```
task LINE_TO_CHAR is  
  entry PUT_LINE (L : in LINE);  
  entry GET_CHAR (C : out CHARACTER);  
end;
```

```
task body LINE_TO_CHAR is  
  BUFFER : LINE;  
begin  
  loop  
    accept PUT_LINE (L : in LINE) do  
      BUFFER := L;  
    end PUT_LINE;  
    for I in BUFFER'RANGE loop  
      accept GET_CHAR (C : out CHARACTER) do  
        C := BUFFER(I);  
      end GET_CHAR;  
    end loop;  
  end loop;  
end LINE_TO_CHAR;
```

USER TASKS

task PRODUCE.LINE;

task CONSUME.CHAR;

task body PRODUCE.LINE is

MY.LINE : LINE;

begin

loop

-- fill MY.LINE from somewhere

LINE.TO.CHAR.PUT_LINE(MY.LINE);

end loop;

end;

task body CONSUME.CHAR is

MY.CHAR : CHARACTER;

begin

loop

LINE.TO.CHAR.GET_CHAR(MY.CHAR);

-- dispose of MY.CHAR;

end loop;

end;

DELAY STATEMENT

delay 3.5;

suspends the task for at least the duration indicated

units are seconds

the expression is of the predefined fixed point type DURATION

For clarity one may write

SECONDS : constant := 1.0;

MINUTES : constant := 60.0;

and then

delay 2.0 * MINUTES + 45.0 * SECONDS;

SELECT STATEMENT

waiting for one of several alternatives

```
task PROTECTED_VARIABLE is  
  entry READ(V : out ELEM);  
  entry WRITE(E : in ELEM);  
end;
```

```
task body PROTECTED_VARIABLE is  
  VARIABLE : ELEM := INITIAL_VALUE;  
begin  
  loop  
    select  
      accept READ(V : out ELEM) do  
        V := VARIABLE;  
      end;  
    or  
      accept WRITE(E : in ELEM) do  
        VARIABLE := E;  
      end;  
    end select;  
  end loop;  
end PROTECTED_VARIABLE;
```

SELECT WITH GUARDS

task BUFFERING is

entry READ (V : out ELEM);

entry WRITE (E : in ELEM);

end;

task body BUFFERING is

BUFFER : array (1..N) of ELEM;

I, J : INTEGER range 1..N := 1;

COUNT : INTEGER range 0..N := 0;

begin

loop

select

when COUNT > 0 =>

accept READ (V : out ELEM) do

V := BUFFER(J);

end;

J := (J mod N) + 1; COUNT := COUNT - 1;

or

when COUNT < N =>

accept WRITE (E : in ELEM) do

BUFFER(I) := E;

end;

I := (I mod N) + 1; COUNT := COUNT + 1;

end select;

end loop;

end BUFFERING;

PACKAGING A TASK

```
package READER_WRITER is  
  procedure READ (X : out ELEM);  
  procedure WRITE (X : in ELEM);  
end;
```

```
package body READER_WRITER is  
  VARIABLE : ELEM := INITIAL-VALUE;  
  
  task CONTROL is  
    entry START; entry STOP;  
    entry WRITE (X : in ELEM)  
  end;  
  
  ...  
  
  procedure READ (X : out ELEM) is  
  begin  
    CONTROL.START; X := VARIABLE;  
    CONTROL.STOP;  
  end;  
  
  procedure WRITE (X : in ELEM) is  
  begin  
    CONTROL.WRITE(X);  
  end;  
  
end;
```

enforcing a protocol

CONTROLLING READERS AND WRITERS

```
task body CONTROL is  
  READERS : INTEGER := 0;  
begin  
  loop  
    select  
      accept START;  
      READERS := READERS + 1;  
    or  
      accept STOP;  
      READERS := READERS - 1;  
    or  
      when READERS = 0 =>  
        accept WRITE(E: in ELEM) do  
          VARIABLE := E;  
        end;  
      end select;  
    end loop;  
end;
```

SELECT WITH ELSE PART

```
select
|
|   when WRITE'COUNT = 0 =>
|       accept START;
|       READERS := READERS + 1;
|
|   or
|
|       accept STOP;
|       READERS := READERS - 1;
|
|   or when READERS = 0 =>
|       accept WRITE (E : in ELEM) do
|           VARIABLE := E;
|       end;
|       loop
|           select
|               |
|               |   accept START;
|               |   READERS := READERS + 1;
|               |
|               |   else
|               |       exit;
|               |   end select;
|           end loop;
|   end select;
```

- new reader only accepted in the absence of writer
- after a writer, accept any waiting readers

TIMED AND CONDITIONAL ENTRY CALLS

timed entry call

```
select  
|   SERVER.REQUEST(SOME_DATA);  
or  
|   delay 45.0 * SECONDS;  
|   -- what to do if service has  
|   -- not started within delay (server  
|   -- too busy)  
end select;
```

conditional entry call

```
select  
|   SERVER.REQUEST(SOME_DATA);  
else  
|   -- do something else if busy  
|   -- server cannot accept request  
|   -- immediately  
end select;
```

load control

TERMINATE ALTERNATIVE

```
task type RESOURCE is  
  entry SEIZE;  
  entry RELEASE;  
end;
```

```
task body RESOURCE is  
  BUSY : BOOLEAN := FALSE;  
begin  
  loop  
    select  
      when not BUSY =>  
        accept SEIZE do BUSY := TRUE; end;  
      or  
        accept RELEASE do BUSY := FALSE; end;  
      or  
        when not BUSY => terminate;  
    end select;  
  end loop;  
end;
```

TASK CLASSIFICATION

two extremes

the server

calls no other task

its body is often of the form

```
loop
|
|  select
|  |
|  |  ...
|  |  end select;
|  end loop;
```

READER_WRITER

the user

has no entries PRODUCE_LINE

intermediate cases

TASK CLASSIFICATION

when a server supports several users:

- no special requirement because the services do not interact
use a package look up table
- mutual exclusion between services
because of possible interference
PROTECTED VARIABLE
use a select statement
- mutual exclusion + temporary unavailability
because conditions cannot be met
BUFFERING
use guards
- mutual exclusion + temporary unavailability
+ control of order (server is a scheduler)
CONTROLLER
use families of entries

EXCEPTION HANDLING

concept of an exception

what can be terminated by an exception

how are exceptions declared

what are handlers

how are exceptions raised

semantics of exceptions their uses

scope and exceptions

suppressing checks

exceptions and tasking

summary of use

CONCEPT OF AN EXCEPTION

situations that prevent completion
of an action

violation of a constraint

a matrix is singular

an exception names such a situation

CONSTRAINT_ERROR

SINGULAR

Raising an exception means telling
the invoker of an action that
this situation has occurred

Handling an exception means executing
some actions in response

EXCEPTION DECLARATIONS

SINGULAR : exception;

ERROR : exception;

Some exceptions are predefined :
declared in STANDARD

CONSTRAINT_ERROR,

NUMERIC_ERROR,

SELECT_ERROR,

STORAGE_ERROR,

TASKING_ERROR : exception;

tasks have an attribute that is an exception

MULTIPLEXER_FAILURE

EXCEPTION HANDLERS

exception handlers may appear
at the end of a sequence
of statements enclosed by begin and end

exception
handlers

```
begin  
    sequence of statements  
  
    exception  
    {  
        when CONSTRAINT.ERROR =>  
            sequence of statements  
        when SINGULAR =>  
            sequence of statements  
        when others =>  
            sequence of statements  
    }  
end;
```

bloc, subprogram body, package body, task body
handlers of the begin-end part

PROPAGATION OF A PREDEFINED EXCEPTION

Consider

$X := A(I);$

▽

in line
expansion

or

hardware
operation

{
 if I in A'RANGE then
 -- Result obtained by
 -- indexing the
 -- array A with I
 else
 raise CONSTRAINT_ERROR;
 end if;
 X := result;

HANDLING AN EXCEPTION

GENERAL CASE

blocks, subprogram, package

no handler =>

execution of unit is abandoned

same exception propagated in caller

a handler exists for the exception =>

execution of the handler terminate

the execution of the unit

TASKS

no implicit propagation

HANDLING AN EXCEPTION

the handlers apply to the
begin-end part

hence an exception in the
declarative part of a block
subprogram or package is
always propagated

library units : abandon

TERMINATING CONDITION

- Exceptions are "terminating" conditions

```
function DIVIDE (U,V: REAL) return REAL is  
begin  
    return U/V;  
exception  
    when NUMERIC-ERROR  $\Rightarrow$  return REAL'LARGE;  
end;
```

RETRYING AN OPERATION

```
for I in 1..10 loop  
  begin  
    READ_TAPE (BLOCK);  
    exit;  
  exception  
    when TAPE_ERROR =>  
      if I = 10 then  
        raise MALFUNCTION;  
      else  
        BACKSPACE;  
      end if;  
    end;  
end loop;
```

EXCEPTIONS AND TASKING

ASYNCHRONOUS INTERVENTIONS

raise T'FAILURE;

T is allowed last wishes

abort T;

terminate unconditionally !

in either case "partners" of T receive
the exception TASKING_ERROR

example of termination sequence :

raise T'FAILURE;

delay 20.0*SECONDS;

abort T;

Abnormal termination

EXCEPTIONS IN COMMUNICATIONS

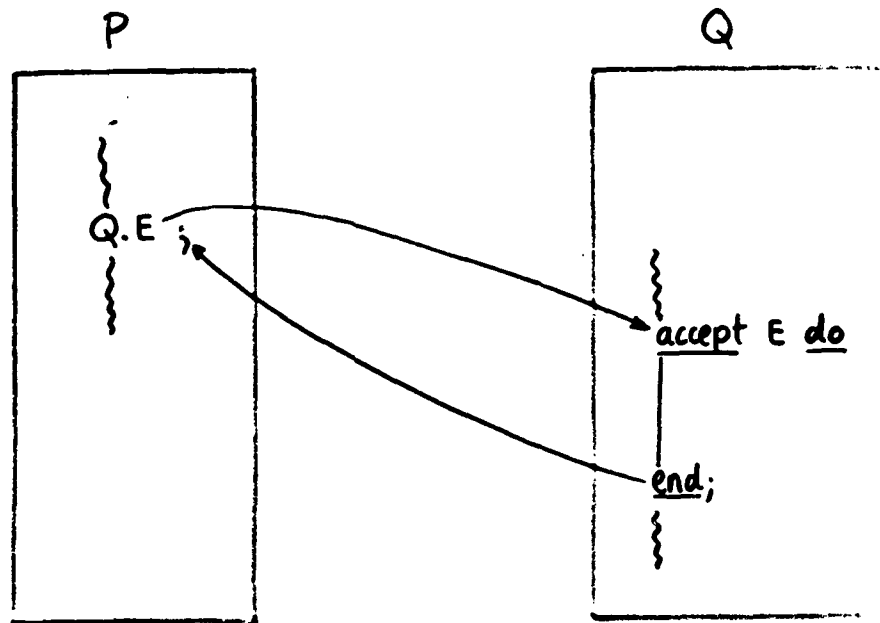
Calling an entry of a
terminated task

the calling task receives
the exception TASKING_ERROR

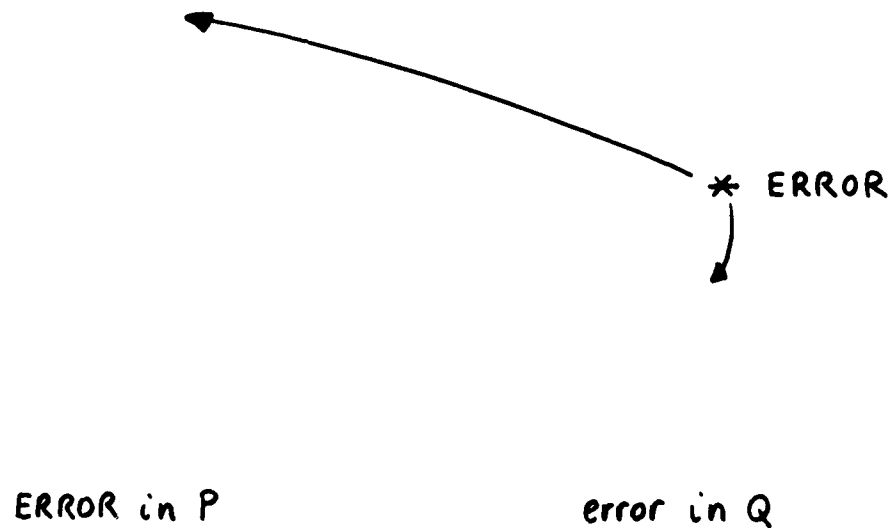
Exceptions occurring during a rendezvous

- during execution of the accept statement
- the called task dies
- the caller dies

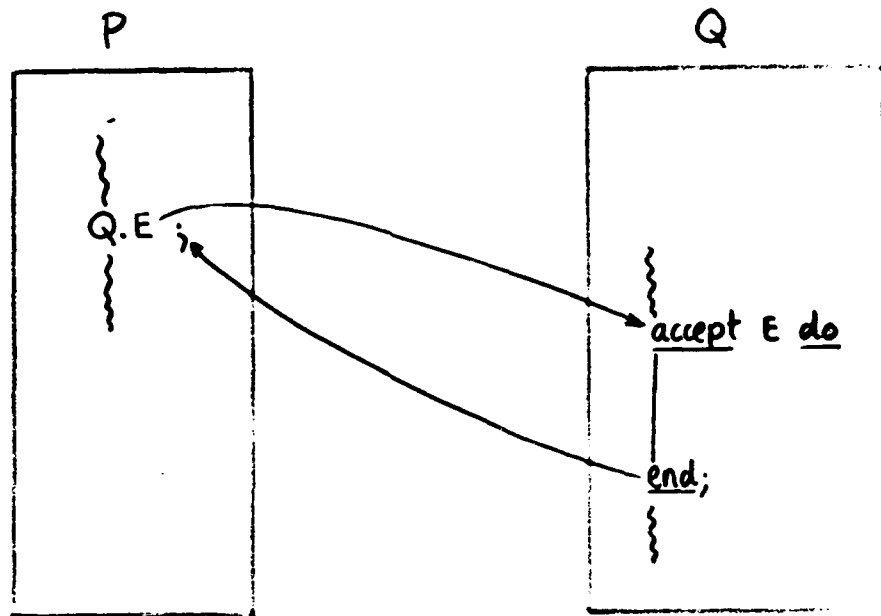
RENDEZVOUS



the exception `ERROR` is raised or
propagated within the `accept` statement

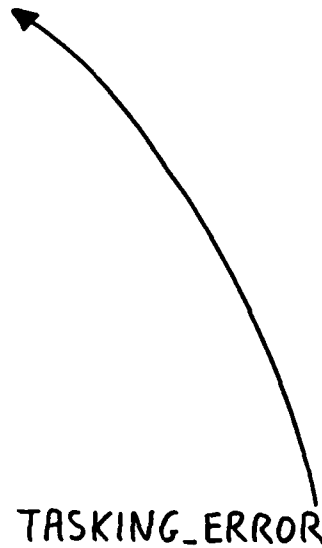


RENDEZVOUS



CALLED TASK

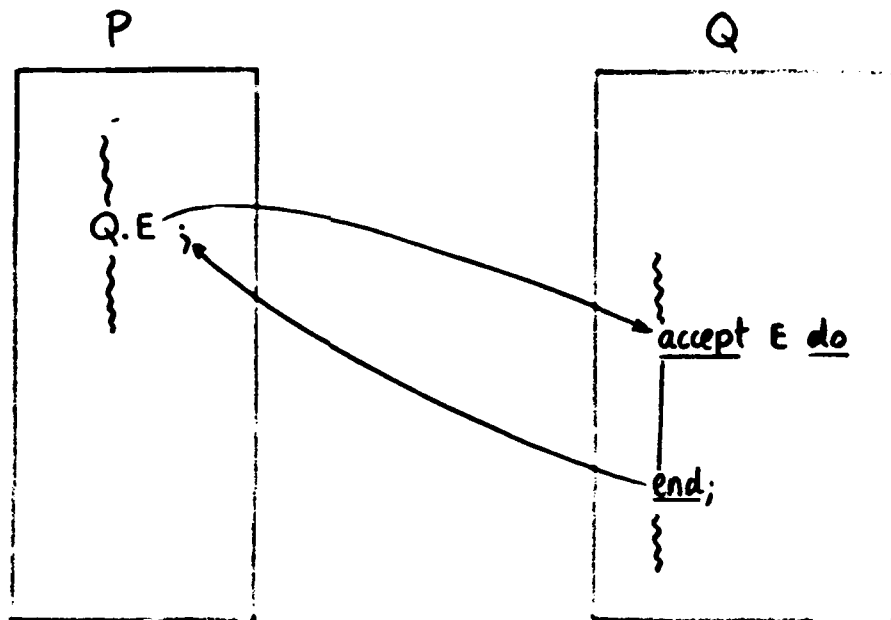
- receives a FAILURE
- is abnormally terminated



TASKING_ERROR

FAILURE is not propagated to calling task

RENDEZVOUS



CALLER DIES

The server completes the rendezvous

The server is unaffected

SUPPRESSING CHECKS

pragma SUPPRESS (RANGE_CHECK);

pragma SUPPRESS (INDEX_CHECK, ON => TABLE);

the pragma may appear in the declarative part of a unit and applies to the end of the unit

checks are not required

but they may be performed nevertheless

hence the corresponding expressions may still occur

they may also be raised explicitly or propagated

SUMMARY OF USE

to regain control when attempted actions cannot be completed

the invoker of the action is given the opportunity to perform appropriate actions

abandon

retry

use an alternative approach

clean-up (LAST WISHES)

continue if appropriate

MR. WILLIAM CARLSON'S CLOSING REMARKS

A frequent complaint that we hear about the software business is that we are still a cottage industry. We are like shoemakers who do not do a very good job providing their own children with shoes. Most software projects use relatively few and primitive tools, and in general software tools are less impressive than the tools that people in the computer aided design business or production control have.

The few programming environments that provide a significant number of productivity enhancing tools are built around a single common language.

I've heard various theories about how to invest in software tools without having a standard language. In fact, many of you probably have heard me talk about the benefits of computer networking and the possibility of having tools written in several different languages which communicate over a network. I can tell you for myself that I was doing that because there wasn't a common language and I was looking for some way to get around that.

Having a common language makes the job of a tool provider very much easier. Now we have the common language that is going to allow us to invest effectively in software tools, standard reuseable packages, and reuseable components. We can begin to capitalize not only the defense software industry, but the software industry as a whole.

I see this Ada Debut not as the end of a program but really as the beginning. The Services are investing significant dollars in building, compilers, and eventually complete families of tools. We at ARPA are going to be pushing very hard to take advantage of having this common language.

I look forward to the 1980's and the benefits that are going to come to us from having a common medium of expression. I'm sure you share my enthusiasm for the outstanding job the Language Design Team has done.

I want to close by thanking Jean Ichbiah for the outstanding presentation these two days, and thank you for coming.